Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Rularea QEMU într-un unikernel Unikraft

Conducător Științific:
Conf.dr.ing. Răzvan Deaconescu

Autor:
Maria Sfîrăială

București, 2025

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

DIPLOMA PROJECT

Running QEMU in a Unikraft Unikernel

Scientific Adviser:
Conf.dr.ing. Răzvan Deaconescu

Author:
Maria Sfîrăială

Bucharest, 2025

I am extremely grateful to my project supervisor, Răzvan Deaconescu, for introducing me to the vast world of systems software. I also could have not undertaken this journey without the help of the maintainers and contributors from both Unikraft and Xen communities. Sharing knowledge is, indeed, caring and improving ones community and professional life.

I would be remiss in not mentioning my family and friends for the unconditional support and guidance. Their efforts to share their experiences has tremendously helped with both my morale and motivation.

Abstract

Unikernels represent a novel cloud technology, introduced in the 2010s as a means of providing specialization at low costs. They are a single-purpose operating system, tightly coupled with the library and application code to achieve small boot times, improved security and efficiency. Combined with a powerful hypervisor, unikernels become the answer to modern cloud-based problems: effortless migration, secure compartmentalization that exceeds the one of container's and fast deployment.

We propose a new application to be run in such a VM, QEMU, with the purpose of uniting two worlds, the hypervisor, that uses the QEMU unikernel to emulate drivers and the VM toolkit used to build the specialized kernel.

Contents

List of Figures

List of Tables

List of Listings

Notations and Abbreviations

ELF – Executable and Linkable Format
HVM – Hardware Virtualized Machine
PV – Paravirtualized
PVH – Enhanced PV
Raspi – Raspberry Pi
TPM – Trusted Platform Module
UDK – Unikernel Development Kit
VM – Virtual Machine
VMM – Virtual Machine Monitor

Chapter 1

Introduction

1.1 Context

As interest in cloud technologies grows, fast, performant and efficient deployment of applications becomes the search for gold of the decade. Docker [2] is a product that encapsulate just that: speedy startup times for applications, facile management and portability, however it does not account for extreme isolation and security.

Unikernels, on the other hand, accomplish all the key components with a few drawbacks [3]. They offer a single address space operating system with the target application linked against the key kernel features to result in an image with small attack surface and great performance thanks to hyper-specialization. The resulting binary runs either baremetal or under a hypervisor, in which case the isolation increases further: there have been close to 0 reported incidents of hyperjacking attacks in the last ten years.

Unikraft is a UDK (Unikernel Development Kit) [24] created with performance and curation in mind. It allows users to seamlessly create, build and run their application as an unikernel, either in native or binary compatible mode, in this way addressing the major drawback attached to unikernels: portability. Still eyeing portability, Unikraft makes it easy to run its VMs under a plethora of platforms: QEMU/KVM, Xen in both PV and PVH mode and Firecracker, with work in progress for platforms such as Hyper-V, bhyve and Raspi baremetal.

1.2 Motivation

While Unikraft offers a multitude of applications and libraries ported to work with its abstractions (or lack thereof when it comes to userspace-kernelspace separation) [23] [20], it has yet to provide a library for virtualization and emulation. Ergo, any user that wishes to run QEMU, Xen, VirtualBox, VMware or other virtualization and emulation software under Unikraft, has to manually go through the tedious task of creating a binary or shared library from the source code, run it together with its dependencies in binary compatible mode and hope that the application does not use features unfit for Unikraft. It is obvious that this approach suffers from many pitfalls and leaves room for improvement, mainly in creating glue code for every such application to be run in native mode in the resulting unikernel.

1.3 Objectives

Having ease of build, stability and portability in mind, we propose porting such a virtualization and emulation piece of software to Unikraft, bringing it into the UDK realm together with its library dependencies: QEMU. In order to satisfy the stability and portability invariants, we opted for a native port (as opposed to a binary-compatibility run) because it makes more room for configuring, patching and creating glue code to accommodate Unikraft's many quirks. We set the objective of having an Unikraft unikernel image running QEMU on Xen as a platform, capable of booting and being debugged.

1.4 Background

1.4.1 Unikernels and Unikraft

Unikernels, also known as library operating systems, the successors of exo-kernels, come in various shapes and forms, centered around a philosophy. The major two directions that emerge are concerned, on one hand, with portability, through syscall shims and binary patching, and on the other hand, with rewriting these abstractions. Unikraft is part of the first category, having implemented a powerful syscall shim layer and an ELF loader library [22] capable of linking and running unmodified POSIX compliant binaries (and shared libraries) in the final VM. While running unmodified applications sounds attractive, this comes with two downsides: the UDK must carry over all the abstractions of the Linux kernel and there's little support for running these unmodified binaries on platforms other than QEMU/KVM.

Unfortunately, the real platform winner when it comes to performance is a type-1 hypervisor, such as Xen, for which there is no support in the Unikraft ELF loader library. Xen removes the host OS layer, present in a micro-VM architecture, as pictured in Figure 1.1 and achieves faster results in a paravirtualized context.

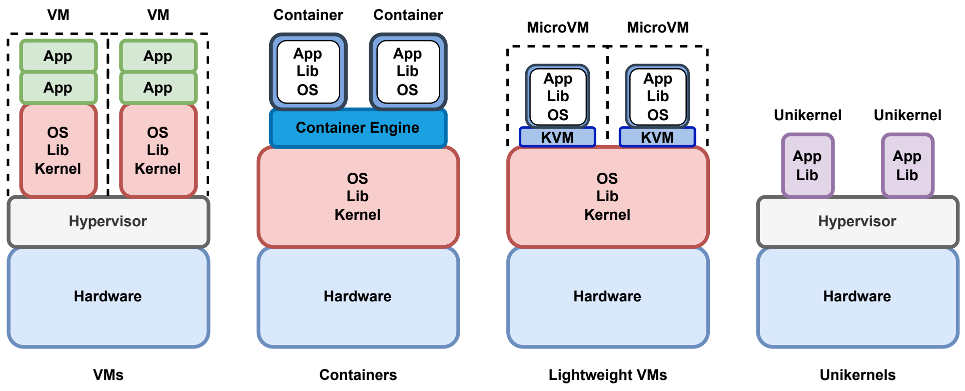


Figure 1.1: VM vs unikernel architecture [6]

1.4.2 Xen and the Art of Virtualization [12]

Xen is a type-1 (also known as native or baremetal) hypervisor, introduced in 2003 in an effort to provide the industry with a VMM capable of managing the notoriously hard to virtualize *x86* architecture. On the *x86* architecture, it occupies ring 0, with the guest kernels being evicted to the unused ring 1, and the applications running in the already established ring 3. It provides a special guest (or domain), named dom0 to fulfill tasks that Xen does not wish to implement, such as device drivers. Therefore, the dom0 guest needs elevated privileges and has to be properly secured. The other domains running under Xen are called domU guests and, depending on the architecture, are aware or not of the virtualized environment they are running in. There are three main types of virtualization provided by Xen [28]:

1. PV (Paravirtualized) in which the guest OS is aware that it is running in a virtualized environment, and was partially rewritten to account for it.
2. HVM (Hardware Virtualized Machine) in which the guest OS takes full advantage of the virtualization extensions introduced in newer processors (VT-x for Intel, and AMD-v for AMD) and runs unmodified.
3. PVH (Enhanced PV) in which the guest OS takes advantage of both paravirtualization and HVM features.

1.4.3 QEMU

QEMU [18] is an open source virtualizer and emulator concerned with full-system emulation, user-mode emulation and virtualization using accelerators such as KVM, Xen, MacOS's Hypervisor Framework (HVF) or Windows Hypervisor Platform (WHPX). One key feature offered by system emulation in QEMU is the ability to emulate devices that range from Virtio (the most performant one as it was originally implemented to work in a paravirtualized context under a hypervisor) [7] to network, USB and NVMe. When it comes to IO emulation, QEMU presents four major building blocks for each device:

1. device front-end, how the guest sees the device.
2. device back-end, how the host does the emulation.
3. device bus, the communication channel to which the device is connected.
4. device pass-through, the ability granted to the emulated device to use the underlying hardware.

Specifically for its device model support, Xen heavily relies on QEMU and has multiple forks of it, some that have been integrated in the upstream version, some that are still pending and used separately [14].

1.4.4 Unikraft, Xen and QEMU

While hardware virtualization extensions made virtualizing *x86* easier, they provided help with only one component, the processor. All other units still needed to be emulated, and Xen accomplished that by integrating QEMU into the software stack with the whole purpose of emulating disk, network, motherboard, and PCI devices.

The first take was to run QEMU in dom0, as showed in Figure 1.2, however that increases the attack surface of the privileged domain, and its task load. The second take, and the more efficient one, was to run QEMU in its own domU, providing every HVM guest with a unique device model. Pictured in Figure 1.3, this hyper-specialized guest is known as a stubdomain (or stubdom)[1] and is currently built as a MiniOS [8] unikernel against an out-of-date QEMU version.

Having started as a Xen Project [25], Unikraft maintains a great relationship with the Xen community and the features that make it a great UDK, ease of apps and libraries porting, POSIX compliance, active members, also call for exchanging the MiniOS device models with Unikraft unikernels running an upstream version of QEMU.

[1]A stubdomain is a specialized system domain running on a Xen host used in order to disaggregate the control domain (dom0) [26].

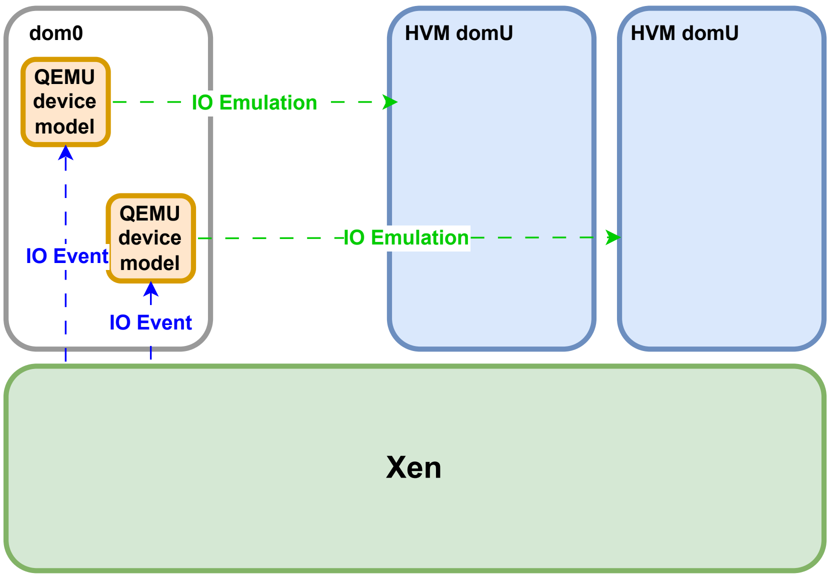


Figure 1.2: QEMU running as a device model in dom0 [13]

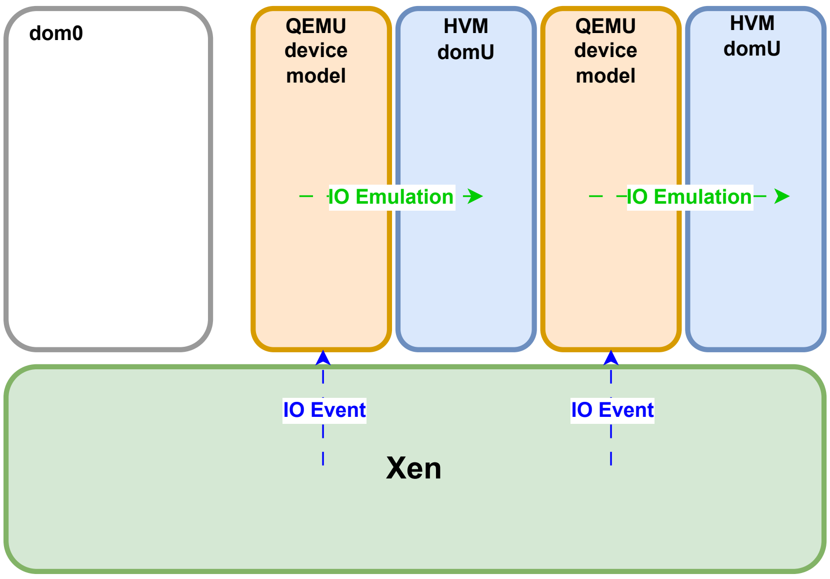


Figure 1.3: QEMU running as a device model in stubdomain [13]

Chapter 2

Related Work

2.1 QEMU Across Platforms

QEMU's accelerator, host operating system and host architecture support is extensive: Table 2.1 illustrates just that, however it does not account for the host operating system running virtualized, let that be a stripped down traditional Linux VM, or a more exotic MiniOS unikernel.

Table 2.1: QEMU support across host OS and architecture [19]

Accelerator	Host OS	Host Architecure
KVM	Linux	AArch64, x64, x86, others
Xen	Linux in dom0	AArch64, AArch32, x64, x86
HVF	MacOS	x64, AArch64
WHPX	Windows	x64, x86
NVMM	NetBSD	x64, x86
TCG	Linux, Windows, MacOS, others	AArch64, AArch32, x64, x86, others

For the purpose of our project, the last two approaches are of interest, since Xen, following the dom0 disaggregation philosophy [27], wishes to spawn an as slim as possible device model stubdomain for each HVM domU that is running at a given moment of time. Naturally, a such QEMU instance runs unprivileged and virtualized, so the usual setup of the process completely differs from the one QEMU was originally built for.

Figure 2.1 shows a basic example of QEMU emulating a disk device doing writes in the two scenarios mentioned above, and its overhead in doing so. One can observe that once with the implementation, the performance also massively differs, therefore we focus on the separate semantics going forward.

2.1.1 MiniOS Unikernels

Spawning a MiniOS unikernel tightly coupled with the QEMU source code was the primary approach when the idea of device model stubdomains came about in Xen, more than 10 years ago: MiniOS provided the necessary minimalistic features that the back then QEMU needed and was lightweight enough to offer great performance, even with multiple HVMs running concurrently. However, the virtualization and emulation sphere faced great improvements over a little period of time, and soon, MiniOS was found to be unable to mirror some of the more advanced features of QEMU. Specifically, the libc MiniOS used, Newlib [4], a library for embedded

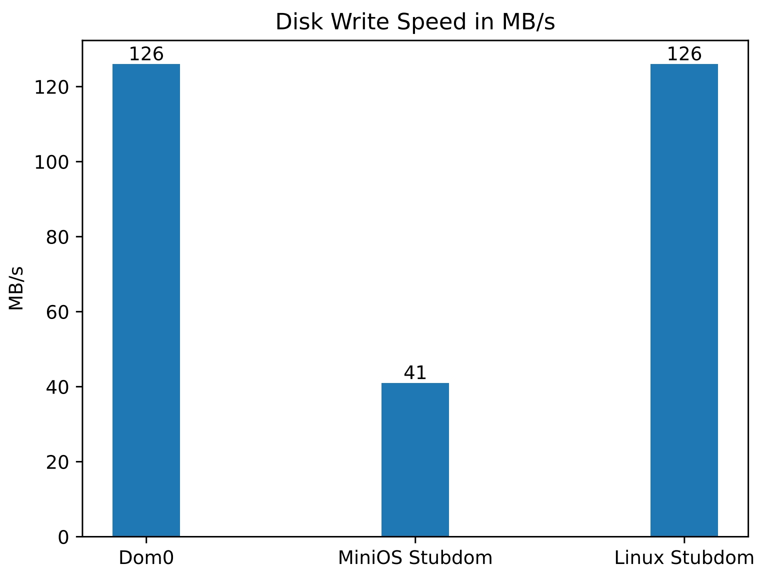


Figure 2.1: Disk writes speed in Linux VMs vs MiniOS unikernels [13]

systems, struggled to supply demanded features, and soon, the MiniOS unikernel stubdomain became an antique relic able to run up to QEMU's 0.97 version.

2.1.2 Linux VMs

As a result of the MiniOS unikernels becoming obsolete, the XenProject community had to devise a new plan. They had two options: either port a more comprehensive libc to be linked against MiniOS (but also work in the core libraries to implement new syscall shims) or run QEMU in a Linux virtual machine, having guaranteed support for the novel features found in the upstream software. The downside of this approach is the heaviness of the VM, which even stripped down occupies way too much memory, but a trade-off had to be made, and XenProject's principal consumer, Qubes OS, adopted the idea and uses it in their ecosystem.

Chapter 3

Solution Design

3.1 Use Cases

Hypothetically, in a UDK competition, Unikraft would win: it presents itself with an active developer base, has regular releases and it is far from the days it used to import MiniOS implementations. It is growing and keeping itself to the industry standard, becoming a great replacement for the MiniOS-based unikernels currently used as stubdomains by Xen. The long awaited decision comes as the 0.97 QEMU version linked against the MiniOS kernel in the stubdomains has lost its purpose due to obsolescence[2] and has become unusable in modern contexts, with modern hardware emulation standards.

By porting QEMU as a library inside Unikraft, we aim to provide a slim, up to date unikernel image running QEMU and an easy manner of bumping its version, through Unikraft's versatile make-based build system. We strive to offer a device model stubdomain that is both slim, easy to configure and specialized to Xen's needs.

Compared to the obsolete MiniOS-based stubdom and the stripped down Linux image that came about as its substitute, an Unikraft unikernel encapsulates the best of both worlds: compact memory footprint, as a result of specialization, and an upstream version of QEMU.

3.1.1 Main Use Case

Porting QEMU as a library in Unikraft benefits both Unikraft and Xen, firstly, by expanding the unikernel project reach and secondly, by replacing a long outdated piece of software: the MiniOS-based device models. The XenProject community is planing on adopting the technology and expanding it to other MiniOS stubdomains, such as the *Xenstored* [32] one, currently running as a daemon inside dom0, and the virtual TPM stubdom [31], which was long ago marked as stale. Figure 2.1 clearly demonstrates the need to provide XenProject with an unikernel-based device model, a more sophisticated one at that. There is significant room for expanding the project reach across multiple communities, but the most interested one would be XenProject.

3.1.2 Other Use Cases

Other XenProject affiliated communities are also interested in the QEMU library port for Unikraft, namely, Qubes OS [11], an operating system developed with security in mind, running most of its processes as qubes[3] under the Xen hypervisor. Figure 3.1 shows the regular

[2]At the time of writing this paper, the upstream QEMU version is 10.0.0.

[3]The term "qube" was originally invented as a non-technical alternative term to "VM".

OS processes running as VMs and the interaction between them according to the Qubes OS philosophy.

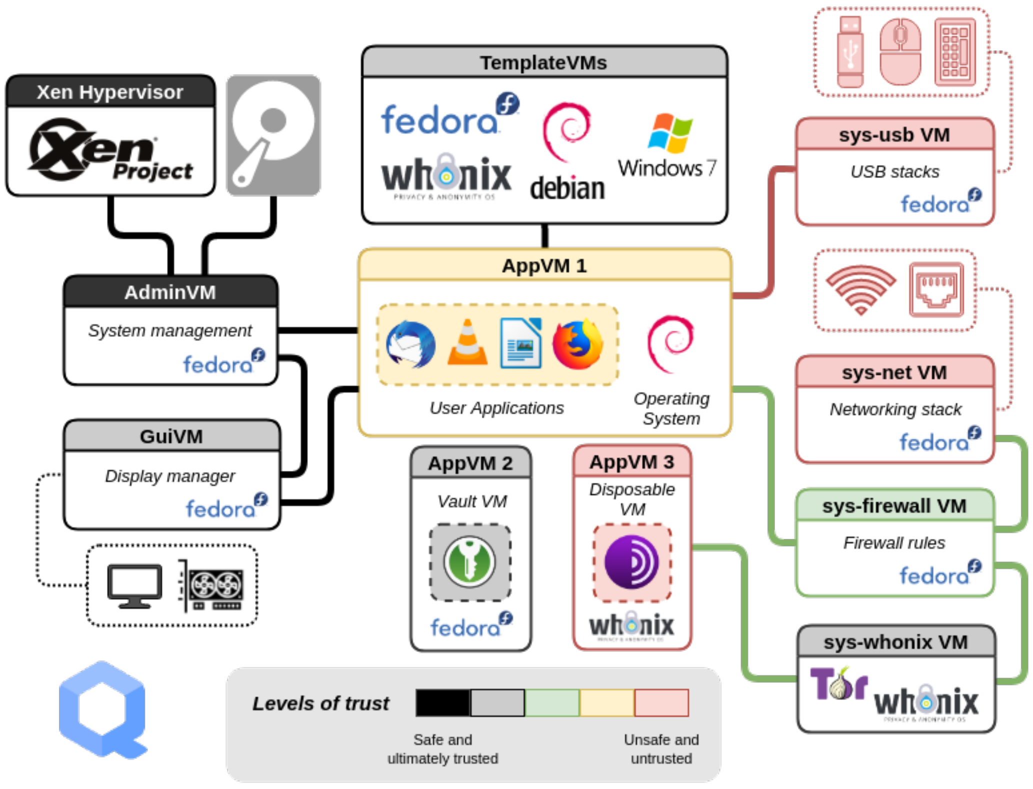


Figure 3.1: Qubes OS architecture [10]

One of these VMs is currently a Linux stubdomain running QEMU [9], with the purpose of maintaining alive a DHCP server and Pulseaudio, but the community is on the lookout to return to an unikernel-based VM as the memory footprint is just too much[1]. QEMU in Unikraft will be able to address the need for lower RAM consumption and, generally, efficient resource usage.

3.2 Building Blocks

Given that QEMU is a machine emulator and virtualizer that aims to support building and executing on a plethora of OSes and for multiple architectures, we focus on configuring the Unikraft QEMU library to be used for the architectures supported by Unikraft and Xen, that being *x64* and *AArch64*. Because there's little interest for *AArch64* in Qubes OS, we set the sensible objective of building the unikernel only for *x64*. Even by reducing the target to only one architecture, the total number of sources that would compile in the final Unikraft VM, from the QEMU side[2] is still around 1000. When it comes to the platform the QEMU unikernel is going to run on, the obvious choice is Xen: it was requested by its community, and it will be used by Qubes OS, a project that constructed their product around the type-1 hypervisor.

We created the QEMU port in Unikraft as an external library, *lib-qemu*, which requires multiple other external libraries: some that need to be freshly ported (*lib-pcre2*, *lib-glib*, to name a few), and some that are already part of the Unikraft ecosystem (*lib-musl*, *lib-zlib* and others). As

[1]Even with a stripped down Linux image, the VM occupies 128-150 MB of RAM.
[2]QEMU is dependent on many external libraries, one of them being Glib

a result, we take advantage of the already implemented robust make-based build system, to integrate the missing library pieces into the big Unikraft picture.

3.3 Architectural Overview

Achieving running QEMU in Unikraft means mostly working on the external libraries mentioned previously: *lib-musl*, *lib-zlib*, *lib-pcre* and *lib-pixman*, which are already ported, and *lib-pcre2*, *lib-glib* and other Xen libraries, to be ported in an effort to support the target *lib-qemu*. Figure 3.2 displays the interaction between these components inside Unikraft, together with two extra features to be achieved in order to have the final unikernel able to boot and make hypercalls.

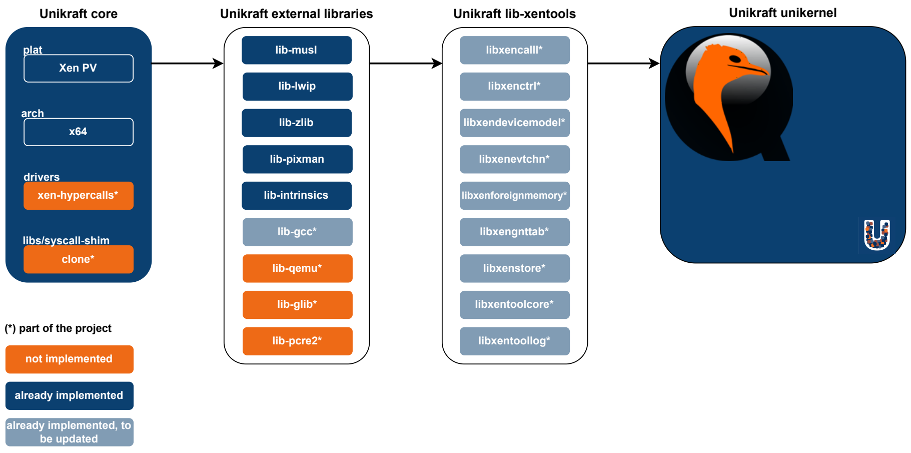


Figure 3.2: *lib-qemu* port architecture

As a result of having the Unikraft QEMU VM running in a Qubes OS context, we have to be aware of the fact that the unikernel has to know how to issue hypercalls, due to the Xen specific libraries that the community is currently using, *libxendevicemodel* and *libxenctrl*, to name a few. Therefore, we include into the implementation an extra component, *xen-hypercalls*, functioning as a driver inside Unikraft core.

What's more, recent work in Unikraft's core library *syscall_shim* hardcoded an assembly prologue in the syscall handling of all system calls that entail saving the context on an auxiliary stack before giving it as an argument, syscalls like *clone*, heavily used by *lib-qemu*. The syscall prologue proves to not be fully agnostic because it uses assembly instructions like *cli*, as shown in Listing 3.1, that work nicely on platforms such as KVM, but not so great on Xen, which has other ways of clearing and setting interrupts (via event channels). Therefore, we are put in the position of refactoring this specific part of *syscall_shim* to create fully portable prologues for all platforms.

```
#define UK_SYSCALL_EXECENV_PROLOGUE_DEFINE(pname, fname, x, ...) \
        long __used \
        pname(UK_ARG_MAPx(x, UK_S_ARG_LONG_MAYBE_UNUSED, __VA_ARGS__)); \
        __asm__ ( \
                ".global␣" STRINGIFY(pname) "\n\t"      \
                "" STRINGIFY(pname) ":\n\t"      \
                "cli\n\t" \
                "/*␣Switch␣to␣the␣per-CPU␣auxiliary␣stack␣*/\n\t" \
                "/*␣AMD64␣SysV␣ABI:␣r11␣is␣scratch␣register␣*/\n\t" \
                "/*␣Our␣stack␣top␣now␣contains␣a␣return␣address\n\t" \
                "␣*␣pushed␣by␣call;␣this␣must␣be␣ignored␣when\n\t" \
                "␣*␣saving␣the␣stack␣pointer␣to␣the␣interrupt\n\t" \
                "␣*␣return␣structure,␣but␣taken␣into␣account␣when\n\t" \
                "␣*␣we␣actually␣return␣execution\n\t" \
                "␣*/\n\t" \
    );
```

Listing 3.1: Non-portable *syscall_ shim* macro that prepends syscalls with execenv properties

Chapter 4

Implementation Details

Porting any piece of software as an external library to Unikraft means molding its build system to Unikraft's rather than the other way around. Unikraft's build lifecycle consists of multiple steps, pictured in Figure 4.1:

1. **Configuring** the unikernel (using Kconfig [1] for ease of selection and dependency check).
2. **Fetching** the remote library code (through complex *make* rules saved in a *Makefile.uk* file).
3. **Preparing** the fetched library code (as stated in *Makefile.uk*).
4. **Compiling** the external library code together with Unikraft's core libraries (using the *make* variables populated by *Makefile.uk*).
5. **Linking** the final unikernel image.

Depending on the library, extra steps must be added: patching, creating glue code in order to ensure (binary) compatibility with Unikraft or auto-generating sources and headers. Nevertheless, the magic happens via two very important files, *Config.uk*, containing the dependencies of the library, and *Makefile.uk*, consisting of the *make* rules that fetch, configure and build the right sources.

Figure 4.2 lists all generated files, and their corresponding core and external Unikraft *make* variables holding them in order to consecutively advance through the build cycle until the bootinfo, debug, and run images are created.

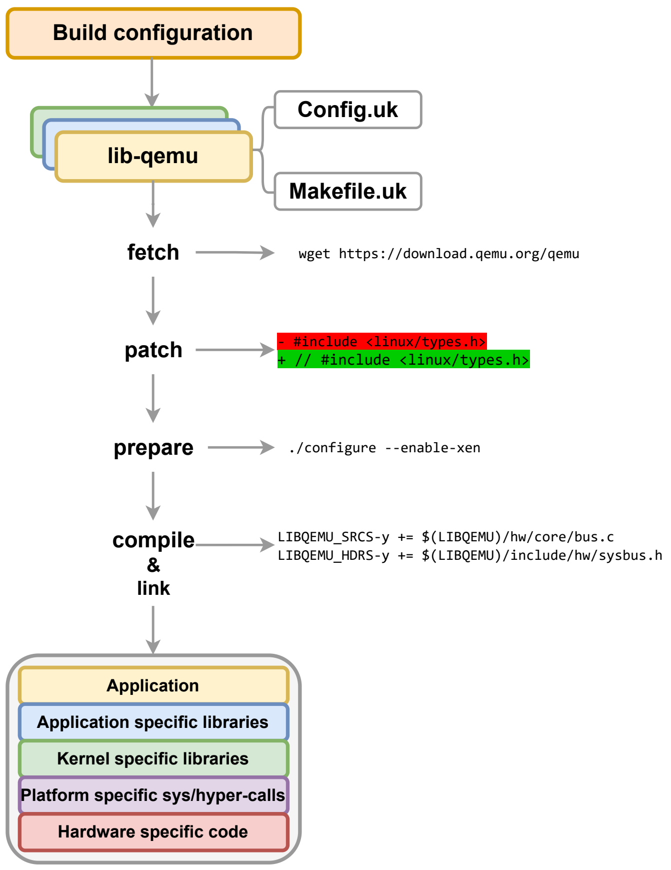
To begin with, *LIB*_ URL* is locally defined for each library involved in the build, and holds the remote address of the library source code we wish to compile.

After fetching and extracting the source code, each source we wish to compile has to manually be added to *LIB*_SRCS-y*, also locally defined in every's library *Makefile.uk*, a build stage closely followed by the compilation of said sources and creation of the *make* variable holding the object files.

If we wish to keep some symbols visible at library scope only, we mask them through an optional *exportsyms.uk* file, populated as shown in Listing 4.1, with the names of the exported outside the library symbols.

```
open
write
read
```

Listing 4.1: *exportsyms.uk*'s exporting symbols



```
wget https://download.qemu.org/qemu
```

```
- #include <linux/types.h>
+ // #include <linux/types.h>
```

```
./configure --enable-xen
```

```
LIBQEMU_SRCS-y += $(LIBQEMU)/hw/core/bus.c
LIBQEMU_HDRS-y += $(LIBQEMU)/include/hw/sysbus.h
```

Figure 4.1: Build stages in Unikraft

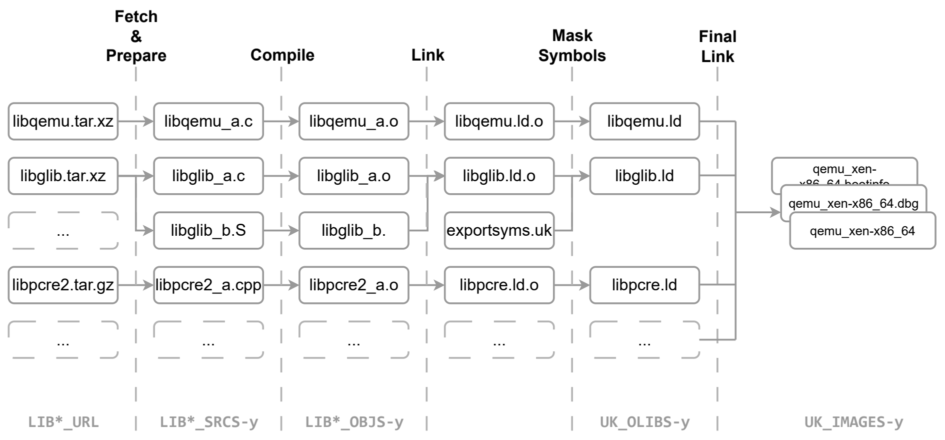


Figure 4.2: Generated files in building Unikraft [21]

The last step before having a bootable unikernel image is linking all the individual libraries together, via the only global *make* variable involved in the process, *UK_ OLIBS-y*. Needless to say, Unikraft supports linking against precompiled static libraries, through *UK_ ALIBS-y*, but that rarely happens, due to the prerequisite of patching and providing glue code.

The external libraries we ported or updated in an effort to bring QEMU as a library in Unikraft, the libraries that follow all the steps mentioned above, are *lib-qemu*, *lib-glib*, *lib-pcre2*, *lib-xentools* and *lib-gcc*.

4.1 *lib-qemu* Implementation

Bringing QEMU to Unikraft mimicked the usual build lifecycle of a C/C++ open source library, as described before. Firstly, we laid down its dependencies and registered them to the menuconfig system, as pictured in Listing 4.2.

```
config LIBQEMU
        bool "QEMU:␣An␣emulation␣library"
        default n
        select LIBMUSL
        select LIBGLIB
        select LIBZLIB
        select LIBPIXMAN
        select LIBXENTOOLS
        select LIBINTEL_INTRINSICS
        select LIBGCC
```

Listing 4.2: *lib-qemu*'s *Config.uk*

4.1.1 Patching QEMU

Secondly, we registered the library, fetched and patched its code with the necessary changes needed by the Qubes OS community, shown in Listing 4.3.

```
#####################################################################
# Library registration
#####################################################################
$(eval $(call addlib_s,libqemu,$(CONFIG_LIBQEMU)))

#####################################################################
# Sources
#####################################################################
LIBQEMU_VERSION=8.1.2
LIBQEMU_URL=https://download.qemu.org/qemu-$(LIBQEMU_VERSION).tar.xz
LIBQEMU_DIR=qemu-$(LIBQEMU_VERSION)/
LIBQEMU_PATCHDIR=$(LIBQEMU_BASE)/patches

$(eval $(call fetch,libqemu,$(LIBQEMU_URL)))
$(eval $(call patch,libqemu,$(LIBQEMU_PATCHDIR),$(LIBQEMU_DIR)))
```

Listing 4.3: *lib-qemu*'s registration, fetching and patching from *Makefile.uk*

4.1.2 Configuring QEMU

Thirdly, we took into account the use case for which QEMU is built in Qubes OS and configured it accordingly, keeping in mind that an unikernel acting as a device model reaches peek performance if it is as slim as possible. As a result, many of QEMU's unnecessary features related to IO emulation can be seen as disabled in Listing 4.4. The little features we kept enabled are concerned with Xen, that we choose as an accelerator, and wiht the target architecture, *x64*.

```
##################################################################
# QEMU prepare
##################################################################
# Run ./configure
$(LIBQEMU_BUILD)/.configured: $(LIBQEMU_BUILD)/.prepared
	$(call verbose_cmd,CONFIG,libqemu: $(notdir $@), \
	cd $(LIBQEMU) && ./configure \
	--cxx=/non-existent \
	--disable-attr \
	--disable-auth-pam \
	--disable-bochs \
	--disable-brlapi \
	--disable-bzip2 \
	--disable-cap-ng \
	--disable-cloop \
	--disable-cocoa \
	--disable-coroutine-pool \
	--disable-crypto-afalg \
	--disable-curl \
	--disable-curses \
	--disable-dmg \
	--disable-docs \
	--disable-gcrypt \
	--disable-glusterfs \
	--disable-gnutls \
	--disable-gtk \
	--disable-guest-agent \
	--disable-hax \
	--disable-kvm \
	--disable-libiscsi \
	--disable-libnfs \
	--disable-libssh \
	--disable-linux-aio \
	--disable-live-block-migration \
	--disable-lzo \
	--disable-netmap \
	--disable-nettle \
	--disable-numa \
	--disable-opengl \
	--disable-parallels \
	--disable-qcow1 \
	--disable-qed \
	--disable-qom-cast-debug \
	--disable-rbd \
	--disable-rdma \
	--disable-replication \
	--disable-sdl \
	--disable-seccomp \
	--disable-slirp \
	--disable-smartcard \
	--disable-snappy \
	--disable-spice \
	--disable-spice \
        --disable-tcg \
        --disable-tools \
        --disable-tpm \
        --disable-usb-redir \
        --disable-vde \
        --disable-vdi \
        --disable-vhost-crypto \
        --disable-vhost-net \
        --disable-vhost-user \
        --disable-virglrenderer \
        --disable-virglrenderer \
        --disable-virtfs \
        --disable-vnc \
        --disable-vte \
        --disable-vvfat \
        --disable-werror \
        --enable-pie \
        --enable-rng-none \
        --enable-trace-backends=log \
        --enable-xen \
        --enable-xen-pci-passthrough \
        --prefix=/usr \
        --target-list=x86_64-softmmu \
        --without-default-features)
```

Listing 4.4: *lib-qemu*'s configuration from *Makefile.uk*

4.1.3 Gathering Sources and Headers

Usually, building the unikernel with the target library is a very iterative process because it requires building the unikernel step-by-step, including new files to the build, making adjustments, re-building, etc. This, and the various and quite quirky build systems libraries impose, is why we must first build the soon-to-be-ported library locally, understand the steps it goes through, its dependencies and sources and only then bring it to Unikraft. *lib-qemu* followed no different path from that: we ran the *./configure* script with a bunch of options disabled, in order to get a lean image, and finally, we called *make* to start the compilation. Typically, this step produces output of great importance for the porting work, because it contains compiler options, flags, sources and includes, all which should be transformed into rules in the ported library *Makefile.uk*.

Unfortunately, minimalistic enough, the *make* output provided merely an idea of what source file was compiling at a given moment of time, as captured by Listing 4.5. In consequence, we had to search for the necessary information elsewhere.

```
[1509/1913] Compiling C object qemu-system-x86_64.p/softmmu_main.c.o
```

Listing 4.5: Snippet of the 1913 lines *make* output

The first place we went to find answers was a log file, saved by Meson, with all the compile commands, conveniently named *compile_ commands.json*. It contains a list with all the compilation and linking commands ran during the build, a snippet being shown in Listing 4.6.

```
{
    "directory": "/home/maria/qemu/build",
    "command": "cc␣-m64␣-mcx16␣-Isubprojects/dtc/libfdt/libfdt.a.p␣-Isubprojects/dtc/libfdt␣-I../subprojects/dtc/libfdt␣-fdiagnostics-color=auto␣-Wall␣-Winvalid-pch␣-Werror␣-std=gnu11␣-O2␣-g␣-Wpointer-arith␣-Wcast-qual␣-Wnested-externs␣-Wstrict-prototypes␣-Wmissing-prototypes␣-Wredundant-decls␣-Wshadow␣-DFDT_ASSUME_MASK=0␣-DNO_YAML␣-DNO_VALGRIND␣-D_GNU_SOURCE␣-D_FILE_OFFSET_BITS=64␣-D_LARGEFILE_SOURCE␣-fno-strict-aliasing␣-fno-common␣-fwrapv␣-fPIE␣-MD␣-MQ␣subprojects/dtc/libfdt/libfdt.a.p/fdt.c.o␣-MF␣subprojects/dtc/libfdt/libfdt.a.p/fdt.c.o.d␣-o␣subprojects/dtc/libfdt/libfdt.a.p/fdt.c.o␣-c␣../subprojects/dtc/libfdt/fdt.c",
    "file": "../subprojects/dtc/libfdt/fdt.c",
    "output": "subprojects/dtc/libfdt/libfdt.a.p/fdt.c.o"
}
```

Listing 4.6: Snippet of an element saved in the array of the compile_commands.json file

Having this array at hand, we were able to parse it, create the necessary source paths and register them in their subsequent sub-library. The headers were parsed by opening each source file and extracting the includes, but unfortunately that wasn't robust enough, as headers can include other headers and so on. We had to find a better solution.

The second approach appeared to be better in the sense that all source and headers to be introduced into the Unikraft build system were found in the *.d* files generated at compile time. Handily, each QEMU sub-library had these dependency files saved separately, so parsing them and producing each sub-library's *Makefile.uk* by applying the Listing B.1 script was greatly eased.

With source and header files introduced to the build system, we started compiling the library. It was no surprise finding out that the original QEMU build system, with all of its many optional configurations disabled, was still too bloated. This was due to the on-cascade selection of configs done by QEMU's *minikconf.py* script, concerned with parsing *.mak* files and checking the dependencies between various configurations. As a result, we patched the original QEMU build system to skip this step and provided our own target, host and devices config headers.

4.1.4 Compartmentalizing *lib-qemu*

Because we are concerned with compartmentalization, we made the decision that *lib-qemu* should be organized in sub-libraries, due to the sheer number of files. Each sub-library has its own *Makefile.uk* which further registers its sources and headers to the build system via *Makefile.rules* as shown in Listing A.1. What's more, *lib-qemu*'s sub-libraries should also be included into the main *Makefile.uk* in order to benefit from the rules defined by *Makefile.rules* as pictured by Listing 4.7.

```
# Additional macros for qemu sub-libraries
include $(LIBQEMU_BASE)/Makefile.rules

################################################################
# QEMU code -- one external Makefile per sub-lib
################################################################
include $(LIBQEMU_BASE)/Makefile.uk.qemu.authz
include $(LIBQEMU_BASE)/Makefile.uk.qemu.block
include $(LIBQEMU_BASE)/Makefile.uk.qemu.blockdev
include $(LIBQEMU_BASE)/Makefile.uk.qemu.chardev
include $(LIBQEMU_BASE)/Makefile.uk.qemu.common
include $(LIBQEMU_BASE)/Makefile.uk.qemu.crypto
include $(LIBQEMU_BASE)/Makefile.uk.qemu.event-loop-base
include $(LIBQEMU_BASE)/Makefile.uk.qemu.fdt
include $(LIBQEMU_BASE)/Makefile.uk.qemu.gdb_softmmu
include $(LIBQEMU_BASE)/Makefile.uk.qemu.gdb_user
include $(LIBQEMU_BASE)/Makefile.uk.qemu.hwcore
include $(LIBQEMU_BASE)/Makefile.uk.qemu.io
include $(LIBQEMU_BASE)/Makefile.uk.qemu.migration
include $(LIBQEMU_BASE)/Makefile.uk.qemu.qemuutil
include $(LIBQEMU_BASE)/Makefile.uk.qemu.qemu-x86_64-softmmu
include $(LIBQEMU_BASE)/Makefile.uk.qemu.qmp
include $(LIBQEMU_BASE)/Makefile.uk.qemu.qom
include $(LIBQEMU_BASE)/Makefile.uk.qemu.qos
```

Listing 4.7: *lib-qemu*'s registration of sub-libraries from *Makefile.uk*

4.1.5 Auto-generating Sources and Headers

Nevertheless, we collected and auto-generated sources and files as seen in QEMU's Ninja/Meson-based original build system. This is captured in Listing 4.8.

```
#####################################################################
# QEMU prepare
#####################################################################
# Auto-generate sources and headers
$(LIBQEMU_BUILD)/.configured: $(LIBQEMU_BUILD)/.prepared
	$(call verbose_cmd,CONFIG,libqemu: $(notdir $@), \
	 $(LIBQEMU_BASE)/helpers/custom_commands.sh $(LIBQEMU) && touch
	     $@)
```

Listing 4.8: *lib-qemu*'s auto-generation of source and header files from *Makefile.uk*

4.1.6 Compiling *lib-qemu*

Compiling *lib-qemu* proved to be a challenge too. QEMU's build system picks up the target operating system on which the final binary will run based on the environmental setup of the machine on which the compilation takes place. For Unikraft, it is highly problematic because it means that very specific Linux features will be expected to be supported. Sources of the fashion of *can_socketcan.c*, as seen in Listing 4.9, therefore become candidates for compilation even though they make use of (problematic for Unikraft) headers. The used headers are pictured in Listing 4.10, and they error out when compiling against the Unikraft codebase simply because Unikraft does not support all Linux kernel features.

```
can_ss.add(when: 'CONFIG_LINUX', if_true: files('can_socketcan.c'))
```

Listing 4.9: Meson assuming Linux specific *can_socketcan.c* source is needed by environmental variable

```
#include "qemu/osdep.h"
#include "qemu/log.h"
#include "qemu/main-loop.h"
#include "qemu/module.h"
#include "qapi/error.h"
#include "chardev/char.h"
#include "qemu/sockets.h"
```

```
#include "qemu/error-report.h"
#include "net/can_emu.h"
#include "net/can_host.h"

#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include "qom/object.h"
```

Listing 4.10: Impossible to compile *can_socketcan.c* source due to specific to Linux headers

Consequently, we had two options going forward: overwrite the target operating system during the configuration step or manually removing all sources introduced into the build system by the set *CONFIG_LINUX* variable. We went for the second option, as the configuration script errors out if there's no traditional operating system detected and modifying that behavior meant patching a piece of software out of the reach for this project.

4.1.7 Linking *lib-qemu*

Having all the object files generated and ready to be linked in the *make* variable *LIBQEMU_-OBJS-y*, we found two other major issues blocking the kernel images from being created.

One problem was having multiple source files with the same name collide when being compiled into the object file. We avoided compiling these duplicates into the same object, by instructing the compiler to prepend a unique prefix to each target, like shown in Listing 4.11.

```
LIBQEMU_HWCORE_SRCS-y += $(LIBQEMU)/hw/core/bus.c|core
LIBQEMU_COMMON_SRCS-y += $(LIBQEMU)/hw/usb/bus.c|usb
```

Listing 4.11: Object prefix for sources with identical names

For each one of the prefixes in Listing 4.11, the resulting object files are *core.bus.o* and *usb.bus.o*, respectively.

The second issue was regarding the *GCC* compiler generating certain arithmetic instructions with the assumption that they will be provided by the automatic link against its lower-level runtime library *libgcc*, assumption that fails in Unikraft's freestanding environment. We bypassed this link error by updating Unikraft's port of *libgcc*, which will be discussed later on in this chapter.

4.2 *lib-glib* Implementation

GLib [16] is a C utility open source library on which QEMU heavily relies for data types, macros and more. Besides using the main GLib library, QEMU also requires porting GModule [17], for dynamically loading modules. Because of the general organization of the library's source code[1], we brought GLib to Unikraft with the option of enabling/disabling GModule as the user sees fit. This is achieved in Listing 4.12 by transforming the library into a mini menuconfig.

```
menuconfig LIBGLIB
        bool "GLib:␣a␣C␣utility␣library"
        default n
        select LIBMUSL
        select LIBPCRE2
```

[1]GLib's tarballs deliver multiple additional libraries: GObject, GModule and GIO

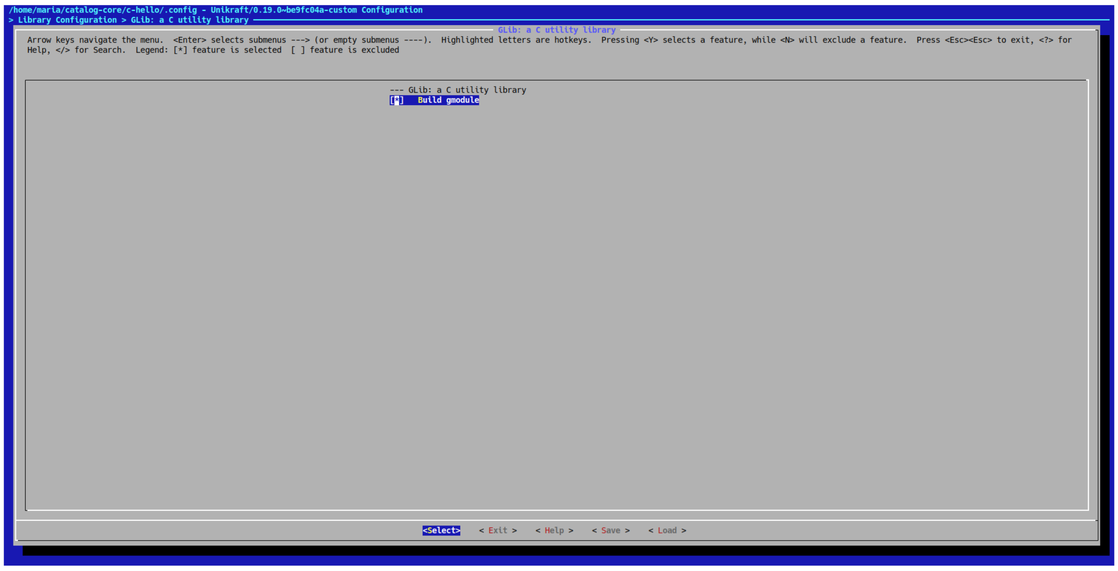
```
        select LIBUKMMAP
        select LIBPOSIX_SYSINFO

if LIBGLIB

config LIBGLIB_GMODULE
        bool "Build␣gmodule"
        default n

endif
```

Listing 4.12: *lib-glib*'s *Config.uk*

4.2.1 Making *lib-glib* a menuconfig

Similarly to *lib-qemu*, GLib exposes an initial configuration step, that sets various macros based on the host system setup, which end up, unfortunately, falsely making assumptions about Unikraft's various capabilities. We bypass this step and include our own headers, with the right modifications.

On the other hand, building GLib does not require all the blocks found in *lib-qemu*: we simply fetch the source code, compile the collected sources (if GModule is selected as pictured in Figure 4.3, some extra files are added — see Listing 4.13) and link the library into the unikernel image.



```
/home/maria/catalog-core/c-hello/.config - Unikraft/0.19.0~be9fc04a-custom Configuration
> Library Configuration > GLib: a C utility library
                                    GLib: a C utility library
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selects a feature, while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is excluded

                                    --- GLib: a C utility library
                                    [*]   Build gmodule

                        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 4.3: Configuring *lib-glib*

```
LIBGLIB_SRCS-$(CONFIG_LIBGLIB_GMODULE) += $(LIBGLIB)/gmodule/gmodule
    .c
LIBGLIB_SRCS-$(CONFIG_LIBGLIB_GMODULE) += $(LIBGLIB)/gmodule/gmodule
    -deprecated.c
```

Listing 4.13: *lib-glib*'s on demand inclusion of GModule sources from *Makefile.uk*

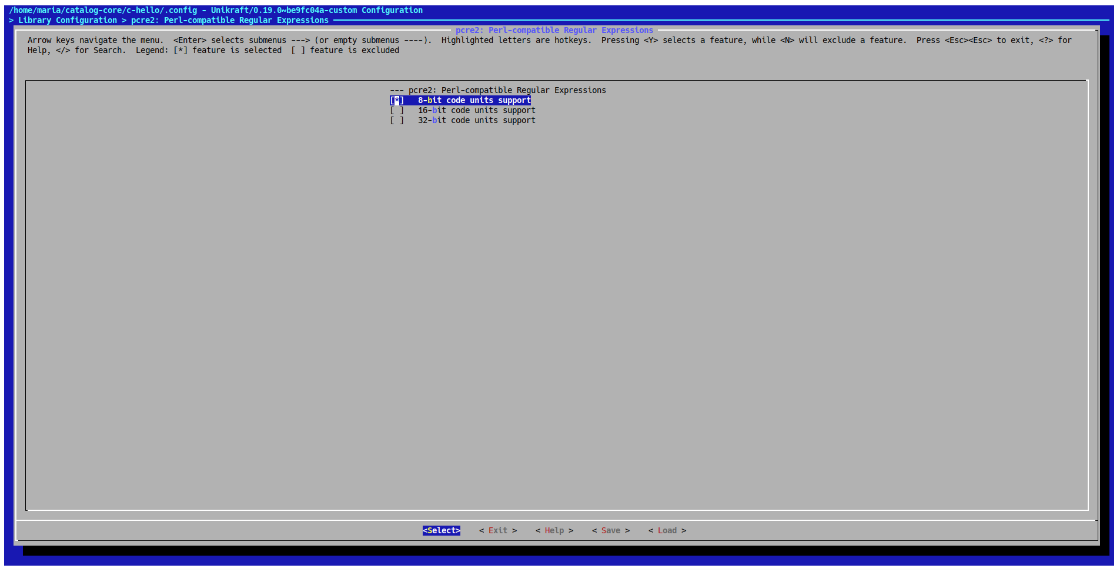
However, GLib is not fully freestanding: without accounting for *lib-musl*, it relies on the PCRE2 library, which should also be introduced to Unikraft.

4.3 *lib-pcre2* Implementation

PCRE2 [5] is a library that offers regular expression pattern matching using Perl like syntax. It is highly configurable and modular and presents itself as one of the easiest library to be ported to Unikraft due to the detailed documentation and design.

4.3.1 *lib-pcre2* with UTF-8,16,32 Encodings

As it can be built for multiple encodings (UTF-8, UTF-16 or UTF-32), we went for a menuconfig approach (shown in Figure 4.4), similar to how we managed *lib-glib*. To achieve this, we passed different preprocessing symbols matching the user's options as seen in Listing 4.14.



```
/home/maria/catalog-core/c-hello/.config - Unikraft/0.19.0~be9fc04a-custom Configuration
> Library Configuration > pcre2: Perl-compatible Regular Expressions
                                   pcre2: Perl-compatible Regular Expressions
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selects a feature, while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
  Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is excluded

                                   --- pcre2: Perl-compatible Regular Expressions
                                   [*]   8-bit code units support
                                   [ ]   16-bit code units support
                                   [ ]   32-bit code units support
```

Figure 4.4: Configuring *lib-pcre2*

```
# Preprocessing symbols
LIBPCRE2_DEFINES-y += -DHAVE_CONFIG_H
LIBPCRE2_DEFINES-$(CONFIG_LIBPCRE2_CODE_UNIT_WIDTH_8) += -
    DPCRE2_CODE_UNIT_WIDTH=8 -DSUPPORT_PCRE2_8
LIBPCRE2_DEFINES-$(CONFIG_LIBPCRE2_CODE_UNIT_WIDTH_16) += -
    DPCRE2_CODE_UNIT_WIDTH=16 -DSUPPORT_PCRE2_16
LIBPCRE2_DEFINES-$(CONFIG_LIBPCRE2_CODE_UNIT_WIDTH_32) += -
    DPCRE2_CODE_UNIT_WIDTH=32 -DSUPPORT_PCRE2_32
LIBPCRE2_CFLAGS-y += $(LIBPCRE2_DEFINES-y)
```

Listing 4.14: *lib-pcre2*'s on demand selection of encodings from *Makefile.uk*

4.4 *lib-xentools* Update

Xen-tools [30] represents a collection of libraries used by domains for interacting with the Xen hypervisor. Some of the functionality it provides is concerned with: XenStore[1], event channels[2], grant tables[3], hypercalls[4], privileged foreign mappings, device models and logging. It is part of

[1]XenStore is a hierarchical namespace (similar to a filesystem) shared between domains.
[2]Event channels are the basic primitive provided by Xen for event notifications, similar to hardware interrupts.
[3]Grant tables provide a generic mechanism to memory sharing between domains.
[4]Hypercalls are a software trap from a domain to the hypervisor.

the Xen hypervisor source tree [29] but acts as a separate subsystem.

Because Unikraft started as a Xen project, it has an early port of Xen-tools that dates back to 2019, the 4.13 release. Nonetheless, that needs to be updated to a newer version in order to match *lib-qemu* and the environment in which the stubdomain will be tested, a 4.18 Xen hypervisor running an Ubuntu 24.04 dom0, ergo, a bump to the 4.18 release for Xen-tools is necessary.

Updating the library meant revisiting the sources that were included, fixing paths for the ones that were moved and removing the ones that were reorganized. An example to follow this idea would be the split of *libxenctrl* into *libxenctrl* and *libxenguest*, that resulted in the decision of not including the latter due to no use-case at hand. The Xen-tools libraries required by *lib-qemu* are *libxencall*, *libxenctrl*, *libxendevicemodel*, *libxenevtchn*, *libxenforeingmemory*, *libgnttab*, *libxenstore*, *libxentoolcore* and *libxentoollog*, all updated to the target 4.18 version through the usual Unikraft build lifecycle.

4.4.1 Renaming Symbols in Xen Platform Code

Without any further modifications, *lib-xentools* successfully links when used together with KVM as a platform. This is not the case when built against Xen platform code and the issue lies in core Unikraft symbol naming: common functions used across Xen driver code clash with identical named symbols in *lib-xentools*. Limiting their scope (through an *exportsyms.uk* file), either at *lib-xentools* or Xen platform code level is not feasible, as both *lib-qemu* and internal drivers act as consumers for these symbols. Therefore, we were left with renaming the colliding symbols inside core Unikraft.

4.5 *lib-gcc* Update

libgcc [15] is a low-level, runtime library, linked against all binaries compiled with *GCC*. It provides functions to match *GCC*'s call generation for code it finds difficult to create inline routines for. For instance, in compiling *lib-qemu*, *GCC* finds it necessary to emit calls to arithmetic functions such as *__udivti3*, *__divti3* or *__popcountdi2*, functions that without a *lib-gcc* port result in an undefined symbol linking error.

Fortunately, Unikraft already provides an early port of *lib-gcc*, consisting of only two sub-libraries from the many *GCC* offers in its source tree, *lib-effi* and *lib-backtrace*. We added a third one, *lib-libgcc*, concerned with solely bringing the routines for integer arithmetic, needed by *lib-qemu*.

Chapter 5

Testing and Evaluation

Bringing new libraries to the Unikraft realm proves to be both important (it improves project reach, portability and impact) and an effort of great detail (it entails working with multiple sources, and headers, build systems and languages). We gathered in Table 5.1 some metrics regarding porting QEMU to Unikraft in order to visualize the magnitude of the project, considering only its external building blocks.

Table 5.1: Unikraft ported libraries code metrics

Library	Number of sources	Lines of code	Lines of patched code
lib-qemu	1091	584091	236
lib-glib	90	131176	0
lib-pcre2	31	59933	0
lib-xentools	59	46414	20
lib-gcc	18	15200	336

For even more context, Figure 5.1 compares the scale of the *lib-qemu* port to the ones of other popular and large Unikraft external libraries. Keeping in mind the fact that *lib-qemu* reached more than half the sources of Unikraft's main standard library, its successful compilation and linking is an achievement on its own.

Putting that aside, in the following chapters we discuss testing and proving that the unikernel image containing QEMU code boots and is debuggable.

5.1 Unikraft Instance Setup

In order to start the QEMU instance, we must first provide an Unikraft main function, called by *ukboot*'s[2] *do_main*. Listing 5.1 showcases a simple QEMU command packaged in the unikernel main routine. In order to reach this point it followed the next path:

1. *_libxenplat_start*, with general processor setup and temporary save of the start info page[3] on the scratch stack (a clean trampoline mechanism to make architecture dependent code easier to be devised).

[2] *ukboot* is the boostrapping library in Unikraft.

[3] A start info page is a read-only general information page mapped early in the boot process into each's domU memory.

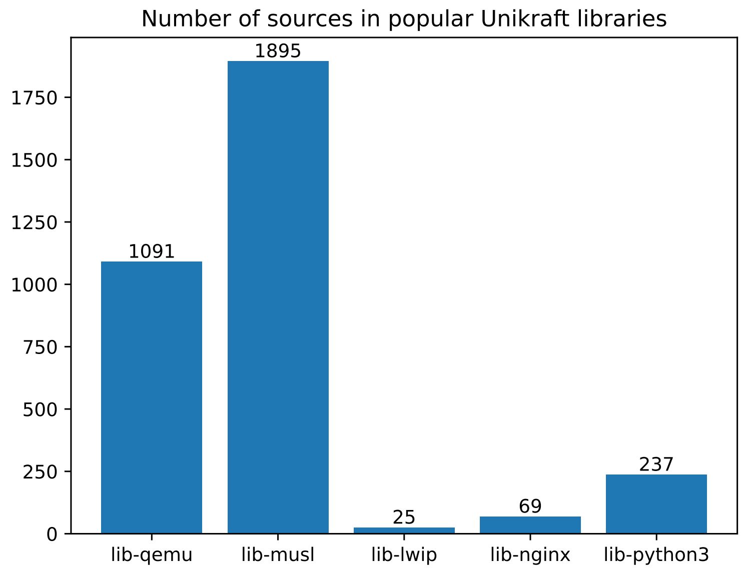


Figure 5.1: Number of sources for popular Unikraft libraries vs *lib-qemu*

2. *_libxenplat_x86entry*, the entry point of the unikernel, when running on Xen; it supplies low-level code for running in a paravirtualized context, it receives the shared info page[1] and sets-up the event channel communication mechanism.
3. *uk_boot_entry*, containing critical setup for the unikernel such as timer and interrupt initialization, Unikraft constructors run and many more.
4. *do_main*, concerned with per app configurations, such as specific constructors and environmental variables.

```c
#include "qemu/osdep.h"
#include "qemu-main.h"
#include "sysemu/sysemu.h"

int qemu_default_main(void)
{
    int status;

    status = qemu_main_loop();
    qemu_cleanup();

    return status;
}

int (*qemu_main)(void) = qemu_default_main;

int main(int argc, char **argv)
{
    qemu_init(2, {"qemu", "--help"});
    return qemu_main();
```

[1]A shared info page is a dynamic general information page mapped into their own memory by each domU.

```
}
```

Listing 5.1: Unikraft main

Up until now, we've covered all the programmatic aspects of the unikernel run, but launching an unikernel under Xen is a bit more complicated as it entails having a dom0 kernel already up and running. We've created the dom0 guest by compiling and installing Xen in a regular Ubuntu 24.04 operating system, followed by reloading the dynamic linker, enabling some Xen specific system services and updating the GRUB. Listing 5.2 contains the Xen related options set before updating the GRUB config that mutate our Linux kernel into a dom0 privileged virtual machine.

```
sudo echo "GRUB_CMDLINE_XEN_DEFAULT=dom0_mem=4096M,max:4096M" >> /
    etc/default/grub
sudo echo "GRUB_CMDLINE_XEN=" >> /etc/default/grub
```

Listing 5.2: GRUB options transforming a regular kernel into a Xen dom0

Listing 5.3 shows the configuration file of the soon-to-be-launched domU and the run command within the dom0. As per this config file, the Unikraft domU will be assigned 1 CPU core, 1000 MBs of memory and the *qemu-hello* name, while the image will be started in PV mode. The actual launcher of the image is a Xen tool, *xl*, created from one of Xen's sub-libraries we mentioned before, *libxenlight*. It is concerned with the management of domUs, its attributions usually referring to creating, pausing, shutdowning and on the fly changing of configs for guest VMs.

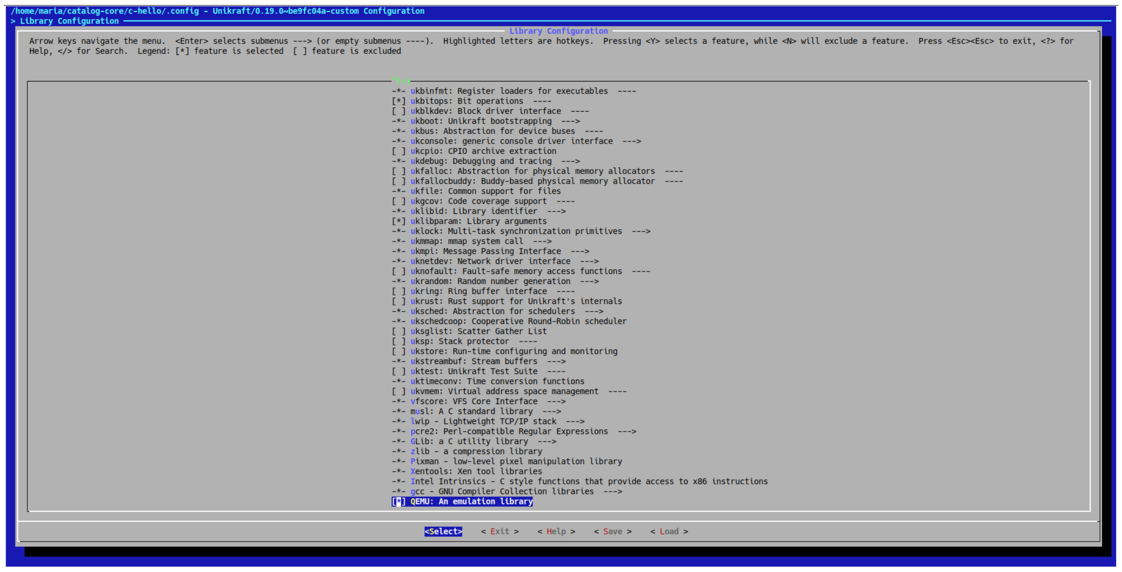
```
maria@frodo:~/catalog-core/qemu-hello$ cat xen.x86_64.cfg
name          = "qemu-hello"
vcpus         = "1"
kernel        = "./workdir/build/qemu-hello_xen-x86_64"
memory        = "1000"
type          = "pv"
maria@frodo:~/catalog-core/qemu-hello$ xl create -c xen.x86_64.cfg
Parsing config from xen.x86_64.cfg
Powered by
o.   .o
Oo   Oo  ___ (_) | __ __  __ _ ' _) :_
oO   oO ' _ `| | |/ /  _)' _` | |_|  _)
oOo oOO| | | | |   (| | | (_) |  _) :_
 OoOoO ._, ._:_:_,\_._,  .__,_:_, \___)
             Pan 0.19.0~be9fc04a-custom
```

Listing 5.3: Running Unikraft command

5.2 Testing Environment Setup

The machine we built and ran *lib-qemu* on is a dom0 Ubuntu 24.04 with an Intel i5-7500, 3.40GHz CPU featuring 4 cores and 11 GBs of allocated memory. Figure 5.2 shows the first step of setting-up our testing environment, meaning the selection of the necessary libraries and options for the fully fledged QEMU instance.

When it comes to the time spent in building all the different libraries that make part of the unikernel running QEMU, Table 5.2 showcases the overall computational work to get the final 12 MBs stripped kernel and 71 MBs debug image.



```
/home/maria/catalog-core/c-hello/.config - Unikraft/0.19.0~be9fc04a-custom Configuration
> Library Configuration
                                   Library Configuration
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selects a feature, while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
 Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is excluded

                                   ^(-)
                                   -*- ukbinfmt: Register loaders for executables  ----
                                   [*] ukbitops: Bit operations  ----
                                   [ ] ukblkdev: Block driver interface  ----
                                   -*- ukboot: Unikraft bootstrapping  --->
                                   -*- ukbus: Abstraction for device buses  ----
                                   -*- ukconsole: generic console driver interface  --->
                                   [ ] ukcpio: CPIO archive extraction
                                   -*- ukdebug: Debugging and tracing  --->
                                   [ ] ukfalloc: Abstraction for physical memory allocators  ----
                                   [ ] ukfallocbuddy: Buddy-based physical memory allocator  ----
                                   -*- ukfile: Common support for files
                                   [ ] ukgcov: Code coverage support  ----
                                   -*- uklibid: Library identifier  --->
                                   [*] uklibparam: Library arguments
                                   -*- uklock: Multi-task synchronization primitives  --->
                                   -*- ukmmap: mmap system call  --->
                                   -*- ukmpi: Message Passing Interface  --->
                                   -*- uknetdev: Network driver interface  --->
                                   [ ] uknofault: Fault-safe memory access functions  ----
                                   -*- ukrandom: Random number generation  --->
                                   [ ] ukring: Ring buffer interface  ----
                                   [ ] ukrust: Rust support for Unikraft's internals
                                   -*- uksched: Abstraction for schedulers  --->
                                   -*- ukschedcoop: Cooperative Round-Robin scheduler
                                   [ ] uksglist: Scatter Gather List
                                   [ ] uksp: Stack protector  ----
                                   [ ] ukstore: Run-time configuring and monitoring
                                   -*- ukstreambuf: Stream buffers  --->
                                   [ ] uktest: Unikraft Test Suite  ----
                                   -*- uktimeconv: Time conversion functions
                                   [ ] ukvmem: Virtual address space management  ----
                                   -*- vfscore: VFS Core Interface  --->
                                   -*- musl: A C standard library  --->
                                   -*- lwip - Lightweight TCP/IP stack  --->
                                   -*- pcre2: Perl-compatible Regular Expressions  --->
                                   -*- GLib: a C utility library  --->
                                   -*- zlib - a compression library
                                   -*- Pixman - low-level pixel manipulation library
                                   -*- Xentools: Xen tool libraries
                                   -*- Intel Intrinsics - C style functions that provide access to x86 instructions
                                   -*- gcc - GNU Compiler Collection libraries  --->
                                   [*] QEMU: An emulation library

                                      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 5.2: Configuring *lib-qemu*

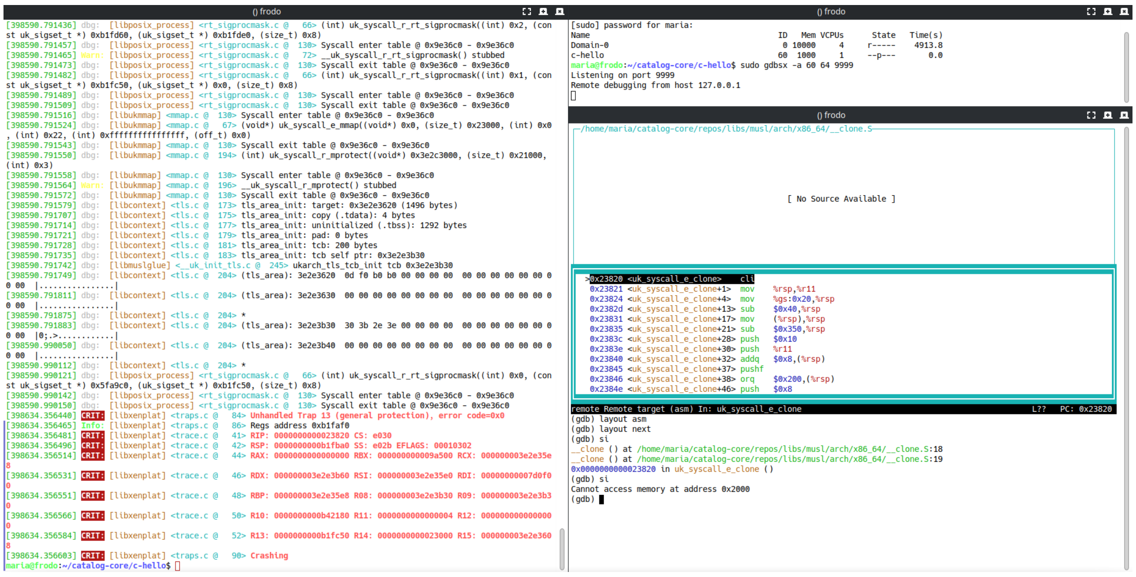
Table 5.2: Unikraft ported libraries build time metrics

Image	Time (in minutes and seconds)
c-hello	0m10,537s
pcre2	0m24,699s
glib	3m38,898s
xentools	2m57,389s
qemu	12m22,125s

Each of the images in Table 5.2 is built as lean as possible, with only the required dependencies and an empty main in order to truthfully reflect the build stage duration.

5.3 Functionality Testing

Discovered in the late phase of the project, the non-portable *syscall_ shim* macro that prepends any syscall with execenv properties we previously mentioned in Listing 3.1, stopped the evolution of the QEMU port. A highly dependent on *clone* support in Xen library, *lib-qemu* spawns multiple threads during application constructor run phase. Figure 5.3 proves the ability to debug the unikernel containing QEMU code, as well as its successful boot (app constructor routines are applied just before reaching the application main - that being the device model entry-point).



```
() frodo
[398590.791436] dbg:  [libposix_process] <rt_sigprocmask.c @   66> (int) uk_syscall_r_rt_sigprocmask((int) 0x2, (con
st uk_sigset_t *) 0xb1fd60, (uk_sigset_t *) 0xb1fde0, (size_t) 0x8)
[398590.791457] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall enter table @ 0x9e36c0 - 0x9e36c0
[398590.791465] Warn: [libposix_process] <rt_sigprocmask.c @   72> __uk_syscall_r_rt_sigprocmask() stubbed
[398590.791473] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall exit table @ 0x9e36c0 - 0x9e36c0
[398590.791482] dbg:  [libposix_process] <rt_sigprocmask.c @   66> (int) uk_syscall_r_rt_sigprocmask((int) 0x1, (con
st uk_sigset_t *) 0xb1fc50, (uk_sigset_t *) 0x0, (size_t) 0x8)
[398590.791489] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall enter table @ 0x9e36c0 - 0x9e36c0
[398590.791509] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall exit table @ 0x9e36c0 - 0x9e36c0
[398590.791516] dbg:  [libukmmap] <mmap.c @  130> Syscall enter table @ 0x9e36c0 - 0x9e36c0
[398590.791524] dbg:  [libukmmap] <mmap.c @   67> (void*) uk_syscall_e_mmap((void*) 0x0, (size_t) 0x23000, (int) 0x0
, (int) 0x22, (int) 0xffffffffffffffff, (off_t) 0x0)
[398590.791543] dbg:  [libukmmap] <mmap.c @  130> Syscall exit table @ 0x9e36c0 - 0x9e36c0
[398590.791550] dbg:  [libukmmap] <mmap.c @  194> (int) uk_syscall_r_mprotect((void*) 0x3e2c3000, (size_t) 0x21000,
(int) 0x3)
[398590.791558] dbg:  [libukmmap] <mmap.c @  130> Syscall enter table @ 0x9e36c0 - 0x9e36c0
[398590.791564] Warn: [libukmmap] <mmap.c @  196> __uk_syscall_r_mprotect() stubbed
[398590.791572] dbg:  [libukmmap] <mmap.c @  130> Syscall exit table @ 0x9e36c0 - 0x9e36c0
[398590.791579] dbg:  [libcontext] <tls.c @  173> tls_area_init: target: 0x3e2e3620 (1496 bytes)
[398590.791707] dbg:  [libcontext] <tls.c @  175> tls_area_init: copy (.tdata): 4 bytes
[398590.791714] dbg:  [libcontext] <tls.c @  177> tls_area_init: uninitialized (.tbss): 1292 bytes
[398590.791721] dbg:  [libcontext] <tls.c @  179> tls_area_init: pad: 0 bytes
[398590.791728] dbg:  [libcontext] <tls.c @  181> tls_area_init: tcb: 200 bytes
[398590.791735] dbg:  [libcontext] <tls.c @  183> tls_area_init: tcb self ptr: 0x3e2e3b30
[398590.791742] dbg:  [libmuslglue] <__uk_init_tls.c @  245> ukarch_tls_tcb_init tcb 0x3e2e3b30
[398590.791749] dbg:  [libcontext] <tls.c @  204> (tls_area): 3e2e3620  0d f0 b0 b0 00 00 00 00  00 00 00 00 00 00 0
0 00  |................|
[398590.791811] dbg:  [libcontext] <tls.c @  204> (tls_area): 3e2e3630  00 00 00 00 00 00 00 00  00 00 00 00 00 00 0
0 00  |................|
[398590.791875] dbg:  [libcontext] <tls.c @  204> *
[398590.791883] dbg:  [libcontext] <tls.c @  204> (tls_area): 3e2e3b30  30 3b 2e 3e 00 00 00 00  00 00 00 00 00 00 0
0 00  |0;.>............|
[398590.990050] dbg:  [libcontext] <tls.c @  204> (tls_area): 3e2e3b40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 0
0 00  |................|
[398590.990112] dbg:  [libcontext] <tls.c @  204> *
[398590.990121] dbg:  [libposix_process] <rt_sigprocmask.c @   66> (int) uk_syscall_r_rt_sigprocmask((int) 0x0, (con
st uk_sigset_t *) 0x5fa9c0, (uk_sigset_t *) 0xb1fc50, (size_t) 0x8)
[398590.990142] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall enter table @ 0x9e36c0 - 0x9e36c0
[398590.990150] dbg:  [libposix_process] <rt_sigprocmask.c @  130> Syscall exit table @ 0x9e36c0 - 0x9e36c0
[398634.356440] CRIT: [libxenplat] <traps.c @   84> Unhandled Trap 13 (general protection), error code=0x0
[398634.356465] Info: [libxenplat] <traps.c @   86> Regs address 0xb1faf0
[398634.356481] CRIT: [libxenplat] <trace.c @   41> RIP: 0000000000023820 CS: e030
[398634.356496] CRIT: [libxenplat] <trace.c @   42> RSP: 0000000000b1fba0 SS: e02b EFLAGS: 00010302
[398634.356514] CRIT: [libxenplat] <trace.c @   44> RAX: 0000000000000000 RBX: 000000000009a500 RCX: 000000003e2e35e
8
[398634.356531] CRIT: [libxenplat] <trace.c @   46> RDX: 000000003e2e3b60 RSI: 000000003e2e35e0 RDI: 00000000007d0f0
0
[398634.356551] CRIT: [libxenplat] <trace.c @   48> RBP: 000000003e2e35e8 R08: 000000003e2e3b30 R09: 000000003e2e3b3
0
[398634.356566] CRIT: [libxenplat] <trace.c @   50> R10: 0000000000b42180 R11: 0000000000000004 R12: 00000000000000
0
[398634.356584] CRIT: [libxenplat] <trace.c @   52> R13: 0000000000b1fc50 R14: 0000000000023000 R15: 000000003e2e360
8
[398634.356603] CRIT: [libxenplat] <traps.c @   90> Crashing
maria@frodo:~/catalog-core/c-hello$
```

```
() frodo
[sudo] password for maria:
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0 10000     4     r-----    4913.8
c-hello                                     60  1000     1     --p---       0.0
maria@frodo:~/catalog-core/c-hello$ sudo gdbsx -a 60 64 9999
Listening on port 9999
Remote debugging from host 127.0.0.1
```

```
() frodo
/home/maria/catalog-core/repos/libs/musl/arch/x86_64/__clone.S

                              [ No Source Available ]

  >0x23820 <uk_syscall_e_clone>     cli
   0x23821 <uk_syscall_e_clone+1>  mov    %rsp,%r11
   0x23824 <uk_syscall_e_clone+4>  mov    %gs:0x20,%rsp
   0x2382d <uk_syscall_e_clone+13> sub    $0x40,%rsp
   0x23831 <uk_syscall_e_clone+17> mov    (%rsp),%rsp
   0x23835 <uk_syscall_e_clone+21> sub    $0x350,%rsp
   0x2383c <uk_syscall_e_clone+28> push   $0x10
   0x2383e <uk_syscall_e_clone+30> push   %r11
   0x23840 <uk_syscall_e_clone+32> addq   $0x8,(%rsp)
   0x23845 <uk_syscall_e_clone+37> pushf
   0x23846 <uk_syscall_e_clone+38> orq    $0x200,(%rsp)
   0x2384e <uk_syscall_e_clone+46> push   $0x8
remote Remote target (asm) In: uk_syscall_e_clone                        L??   PC: 0x23820
(gdb) layout asm
(gdb) layout next
(gdb) si
__clone () at /home/maria/catalog-core/repos/libs/musl/arch/x86_64/__clone.S:18
__clone () at /home/maria/catalog-core/repos/libs/musl/arch/x86_64/__clone.S:19
0x0000000000023820 in uk_syscall_e_clone ()
(gdb) si
Cannot access memory at address 0x2000
(gdb)
```

Figure 5.3: Debugging Unikraft image built with *lib-qemu*

Chapter 6

Conclusion and Further Work

To conclude, we set the goal of having a heavy, huge emulator and virtualizer running in Unikraft, and we delivered just that: QEMU boots, calls its constructors and is debuggable. Unikraft's little support for advanced syscalls on our target platform, Xen, calls for improvement in the portability area, shims and abstraction implementation. Regardless of QEMU's rigid build system, Linux dependency and non-existent freestanding target, we achieved an image that not only competes with *lib-musl* when measuring ported sources and headers, but also links, runs and calls more than 200 application constructors.

As a side effect, we additionally improved native library support: Unikraft will benefit from two freshly ported libraries, *lib-glib* and *lib-pcre2*, and two major updates in *lib-xentools* and *lib-gcc*. This will prove to be of great interest outside *lib-qemu*'s sphere: future users of Unikraft will be welcomed with a UDK capable of natively building virtually any application of their choice.

Considering future work, we set the goal of upstreaming all newly ported or updated libraries involved in our project, *lib-qemu*, *lib-glib*. *lib-pcre2*, *lib-xentools* and *lib-gcc*. Moreover, we plan on rethinking abstractions for *clone*-like syscalls to be fit for all Unikraft platforms. We extend the reach of out efforts to also test and adapt the Qubes OS device model using a stripped down Linux VM to an Unikraft unikernel, with the final aim being updating the stubdomains used by Xen. It entails understanding and modifying their build system to use Unikraft as an unikernel producer and *xl* as an unikernel consumer.

Appendix A

Compartmentalizing lib-qemu Through Makefiles

A.1 Makefile.rules

```
#
# Import a sub-library of qemu
#
# @param $(1)
#   The name of the sub-library within qemu.
# @param $(2)
#   The headers for this sub-library.
# @param $(3)
#   The source files for this sub-library.
#
define _libqemu_import_lib
$(LIBQEMU_BUILD)/include/$(1):
	$(call verbose_cmd,MKDIR,libqemu: $(subst $(LIBQEMU_BUILD),,$(1)): $(notdir $$@), mkdir -p $$@)

# Make a symbolic link of the original header file to a sub-library directory
$(LIBQEMU_BUILD)/include/$(1)/%.h:
	$$(Q)mkdir -p $$(shell dirname $$@)
	$(call verbose_cmd,HOSTLN,libqemu: $(1): $$(subst $(LIBQEMU_BUILD)/include/$(1)/,,$$@), \
		ln -sf $$(subst $(LIBQEMU_BUILD)/include/$(1)/,$(LIBQEMU),$$@) $$@)

# includes for building libqemu
LIBQEMU_$(call uc,$(1))_INCLUDES-y += -I$(LIBQEMU_BUILD)/include/$(1)/src/internal
LIBQEMU_$(call uc,$(1))_INCLUDES-y += -I$(LIBQEMU_BUILD)/include/$(1)/src/$(1)
LIBQEMU_$(call uc,$(1))_INCLUDES-y += -I$(LIBQEMU_BUILD)/include/$(1)/include
LIBQEMU_SRCS-y += $(3)
LIBQEMU_CINCLUDES-y += $$(LIBQEMU_$(call uc,$(1))_INCLUDES-y)
```

```
LIBQEMU_CXXINCLUDES-y += $$(LIBQEMU_$(call uc,$(1))_INCLUDES-y)

# includes for using libqemu
CINCLUDES-$(CONFIG_LIBQEMU) += -I$(LIBQEMU_BUILD)/include/$(1)/
    include
CXXINCLUDES-$(CONFIG_LIBQEMU) += -I$(LIBQEMU_BUILD)/include/$(1)/
    include

# Append the sub library directory to the include path
$(LIBQEMU_BUILD)/.prepared: $(subst $(LIBQEMU),$(LIBQEMU_BUILD)/
    include/$(1),$(2))
endef
```

Listing A.1: Compartmentalizing Makefile (Makefile.rules)

Appendix B

Parsing sources and headers based off dependency files script

B.1 get_sources_headers.py

```
#!/usr/bin/python3

import re
from pathlib import Path

p = Path('.')
origin_p = Path('/home/maria/qemu')

def find_lib(extension):
    lib_dirs = list(p.glob(f'**/*{extension}'))

    for l in lib_dirs:
        files = list(l.glob('**/*.c.o.d'))
        headers = []

        for f in files:
            with open(str(f), "r") as read_file:
                paths = [word for line in read_file for word in line
                    .split() if word != "\\"]
                paths = paths[1:]

                for path in paths:
                    new_p = Path(path)
                    if not new_p.is_absolute():
                        if str(new_p).startswith(".."):
                            new_p = "/home/maria/qemu" / Path(*new_p
                                .parts[1:])
                        else:
                            new_p = "/home/maria/qemu/build" / new_p
                        headers.append(str(new_p))
                    elif origin_p in new_p.parents:
                        headers.append(str(new_p))

            headers = list(set(headers))

        lib = re.search(f'lib(.*){extension}', str(l.name))
        lib = lib.group(1)

        if "fa" in lib:
            pass

        headers_parsed = []
        for h in headers:
            if h.endswith(".c"):
                h = h.replace("/home/maria/qemu", f"LIBQEMU_{lib.
                    upper()}_SRCS-y␣+=␣$(LIBQEMU)")
            else:
                h = h.replace("/home/maria/qemu", f"LIBQEMU_{lib.
                    upper()}_HDRS-y␣+=␣$(LIBQEMU)")
            headers_parsed.append(h)

        headers_parsed.sort()

        with open(f"Makefile.uk.qemu.{lib}", "w+") as write_file:
            ok = False
            for h in headers_parsed:
                if not ok and "SRCS" in h:
                    h = "\n" + h
                    ok = True
                write_file.write(h + "\n")

            write_file.write(f"\n$(eval␣$(call␣_libqemu_import_lib,{
               lib},$(LIBQEMU_{lib.upper()}_HDRS-y),$(LIBQEMU_{lib.
               upper()}_SRCS-y)))\n")

find_lib(".fa.p")
find_lib(".a.p")
```

Listing B.1: Sources and headers parser (get_sources_headers.py)

Bibliography

[1] T. L. K. D. Community. The kconfig build language. https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html.

[2] Inc. Docker. Docker website. https://www.docker.com/.

[3] C. Lupu F. Manco. My vm is lighter (and safer) than your container. SOSP, http://cnp.neclab.eu/projects/lightvm/lightvm.pdf, October 2017.

[4] Red Hat. The newlib homepage. https://sourceware.org/newlib/.

[5] Philip Hazel. Pcre - perl compatible regular expressions. https://www.pcre.org/.

[6] N. Jois. Understanding unikernels. https://nithinjois.com/understanding-unikernels/.

[7] The Linux Kernel. Virtio on linux. https://docs.kernel.org/driver-api/virtio/virtio.html.

[8] MiniOS. Website. https://minios.dev/.

[9] Qubes OS. Linux stubdomain website. https://github.com/qubesos/qubes-vmm-xen-stubdom-linux.

[10] Qubes OS. Qubes os architecture. https://www.qubes-os.org/intro/.

[11] Qubes OS. Website. https://www.qubes-os.org/.

[12] B. Dragovic P. Barham. Xen and the art of virtualization. https://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf, January 2003.

[13] A. Perard. Linux stubdomain. https://xenproject.org/blog/linux-stub-domain/.

[14] A. Perard. Qemu vs qemu-traditional update. https://xenproject.org/blog/qemu-vs-qemu-traditional-update/.

[15] GNU Project. The gcc low-level runtime library. https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html.

[16] The GTK Project. Glib – 2.0. https://docs.gtk.org/glib/.

[17] The GTK Project. Gmodule – 2.0. https://docs.gtk.org/gmodule/.

[18] QEMU. Qemu website. https://www.qemu.org/.

[19] QEMU. Virtualisation accelerators. https://www.qemu.org/docs/master/system/introduction.html#id1.

[20] Unikraft. Unikraft binary-compat application catalog. https://github.com/unikraft/catalog.

[21] Unikraft. Unikraft build process. https://unikraft.org/docs/internals/build-process.

[22] Unikraft. Unikraft elf loader website. https://github.com/unikraft/app-elfloader.

[23] Unikraft. Unikraft native application catalog. https://github.com/unikraft/catalog-core.

[24] Unikraft. Website. https://unikraft.org/.

[25] Unikraft. Xen website. https://xenproject.org/projects/unikraft/.

[26] XenProject. Device model stub domains. https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains.

[27] XenProject. Dom0 disaggregation. https://wiki.xenproject.org/wiki/Dom0_Disaggregation.

[28] XenProject. Understanding the virtualization spectrum. https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum.

[29] XenProject. Xen source tree. https://xenbits.xen.org/gitweb/?p=xen.git;a=tree.

[30] XenProject. Xen toolstack library api/abis. https://xenbits.xen.org/people/liuw/libxenctrl-split/vwip.html.

[31] XenProject. Xen virtual tpm stubdomain. https://xenbits.xen.org/docs/unstable/man/xen-vtpmmgr.7.html.

[32] XenProject. Xenstored documentation. https://xenbits.xenproject.org/gitweb/?p=xen.git;a=blob;f=docs/misc/xenstore.txt;h=4eccbc2f7f2fe46e967413c7da7e825a99f18702;hb=refs/heads/staging.